第12章运算符重载

本章内容

- C++中，可以为类定义运算符，称为运算符重载。
- 恰当地使用运算符，可以简化代码的书写，改善代码的可读性。

使用运算符重载的示例

- 向量的运算

  Class Vector;

  Vector v1,v2;

  Vector v3 = v1 + v2;

  Vector v4 = V1 * V2

  - 比较：
    - Vector v5 = v1.add(v2);
- 虚数、矩阵等函数等数学概念的运算，都可以通过重载运算符改善可读性

使用运算符重载的示例

- 定义IO输入输出
  - cout << “message”
  - 比较：
    - System.out.println(“message”);
- 访问数据表
  - CustomerTable customers;
  - Customer c = customers[“C0001”];
  - 比较：
    - Customer c = customers.findByPK(“C0001”);

运算符重载的两个极端

- 滥用
  - 运算符重载确实是一个有用的工具，但他仅仅只是一种语法上的方便，是函数调用的另一种方式。
  - 只有在能够使得代码更容易书写和阅读时，才考虑重载运算符
    - 一个好的重载方案能够促进代码的书写和理解
- 恐慌
  - 担心所有操作符功能变了

运算符重载

- 运算符的重载从另一个方面体现了OOP技术的多态性，且同一运算符根据不同的运算对象可以完成不同的操作。
- 为了重载运算符，必须定义一个函数，并告诉编译器，遇到这个重载运算符就调用该函数，由这个函数来完成该运算符应该完成的操作。这种函数称为运算符重载函数，它通常是类的成员函数或者是友元函数。运算符的操作数通常也应该是类的对象。

运算符重载

- 定义重载的运算符实际上是定义函数，只是该函数的名字是operator@，其中，@代表了被重载的运算符。

A operator + (A &); //重载了类A的“+”运算符

其中：operator是定义运算符重载函数的关键字，它与其后的运算符一起构成函数名。

运算符重载

- 可以使用函数表示法和运算符表示法来调用它：
  - total = coding.operator+(fixing); // 函数表示法
  - total = coding + fixing; // 运算符表示法
- 这两种表示法都将调用operator+()方法，在运算符表示法中，运算符左侧的对象（ coding ）是调用对象，运算符右边的对象(fixing)是作为参数被传递的对象

运算符重载的语法

- 函数参数表中的参数的个数取决于两个因素：
  - 运算符是一元的（一个参数）还是二元的（两个参数）
  - 运算符被定义为全局函数（参数同上）还是成员函数（一元没有参数，二元一个参数）
  - 一般来说，一元运算符最好被重载为成员函数；对二元运算符最好被重载为友元函数。

运算符重载

- 用成员函数实现运算符的重载时，运算符的左操作数为当前对象，并且要用到隐含的this指针
- 运算符重载函数不能定义为静态的成员函数，因为静态的成员函数中没有this指针。

运算符重载的语法

- 示例：
  - C12: OperatorOverloadingSyntax.cpp

重载运算符的规则

- C++中的大多数运算符都可以重载，但重载时有以下的限制：
  - 不能使用C++中没有定义的运算符，比如试图将”**”定义为求幂；
  - 不能改变运算符参数的个数、运算符的优先级和结合性，也不能改变运算符操作数的语法结构；
  - 运算符重载实质上是函数重载，因此编译程序对运算符重载的选择，遵循函数重载的选择原则；
  - 运算符重载不能改变该运算符用于内部类型对象的含义

重载运算符的规则

- C++中的大多数运算符都可以重载，但重载时有以下的限制：
  - 运算符重载是针对新类型数据的，实际需要对原有运算符进行的适当的改造；
  - 重载的功能应当与原有功能相类似，避免没有目的地使用重载运算符；
  - 重载的运算符的参数不能全部是C++的标准类型，以防止用户修改用于标准类型数据的运算符的性质；

可重载的一元运算符

- 可重载的一元运算符包括：
  - 没有副作用的运算符：+ , - , & , !
  - 有副作用的运算符：++(前缀和后缀), --(前缀和后缀)
- 示例：
  - C12:OverloadingUnaryOperators.cpp

可重载的一元运算符

- 示例中需要注意的内容：
  - 成员运算符和非成员运算符的区别
  - 有副作用的运算符与无副作用的运算符的区别
  - 前缀运算符和后缀运算符的区别
  - 参数和返回值类型的区别

可重载的二元运算符

- 可重载的二元运算符包括：
  - 创建了经修改的新值
    - +, -, *, /, % ,^, &, |, <<, >>
  - 修改左值的值
    - +=, -=, *=, /=, %=, ^=, &=, |=, >>=, <<=
  - 返回真/假的条件表达式
    - ==, !=, <, >, <=, >=, &&, ||

可重载的二元运算符

- 示例：
  - C12:Integer.h
  - C12:Integer.cpp
  - C12:IntegerTest.cpp

  - C12:Byte.h
  - C12:ByteTest.cpp

不可重载的运算符

- C++中绝大部分的运算符允许重载，但是有5个不能重载的运算符。
  - .　　（成员访问运算符）
  - .*　　（成员指针访问运算符）
  - ::　　（域运算符）
  - sizeof　（长度运算符）
  - ?:　　（条件运算符）

重载操作符：参数和返回值

- 对于任何函数参数，如果仅需要从参数中读而不需要改变，默认地应当作为const引用来传递。如果需要改变，则应该作为引用来传递。
- 返回值的类型取决于运算符的具体含义。如果使用该运算符的结果是产生一个新值，就需要产生一个作为返回值的新对象，这个对象通过传值的方式返回。

重载操作符：参数和返回值

- 所有的赋值运算都改变左值。赋值运算符的返回值对于左值应该是非常量引用；
- 对于逻辑运算符，返回一个int或bool类型的值。

重载操作符：参数和返回值

- 返回值优化（return value optimization）
  - 对于return Integer(left.i + right.i)，编译器将进行优化处理，直接将对象创建在外部返回值的内存单元；
  - 因为不是真正创建一个局部对象，所以仅需要一个普通构造函数调用，且不会调用析构函数。效率很高。
    - Integer tmp(left.i + right.i);
    - return tmp;
  - 需要：1)创建tmp对象；2)拷贝构造函数把tmp对象拷贝到外部返回值的存储单元；3)tmp在作用域结尾调用析构函数；

重载操作符：参数和返回值

- 前置和后置（自增自减）运算符
  - ++为前置运算时，它的运算符重载函数的一般格式为：
  - A operator ++(A &a) { ……; }
  - ++为后置运算时，它的运算符重载函数的一般格式为：
  - A operator ++(A &a, int) { ……; }

```
class A
{       int i;
public:public:
        A(int a=0)      { i=a;  }
        friend A operator++(A &a){ a.i++;  return a;}
        friend A operator++(A &a, int n)
        {  A  t;            t.i=a.i;   a.i++;        return t;}
};
void main(void)
{       A a1(10),a2,a3;
        a2=++a1;
        a3=a1++;
}
```

相当于a2=operator++(a1)

相当于a3=operator++(a1,int)

非成员运算符

- 在前面的例子中，有些是成员运算符，有些是非成员运算符。应该如何选择？
  - 原则：如果没有什么差异，尽可能采用成员运算符
- 如果运算符的两侧可能是不同类型的对象，考虑到扩充的可能性，该运算符应该定义为非成员运算符。
- 例：
  - C12: IostreamOperatorOverloading.cpp

非成员运算符

- 基本方针by Murray

Operator	Recommended use
All unary operators	member
= ( ) [ ] –> –>*	*must* be member
+= –= /= *= ^= &= |= %= >>= <<=	member
All other binary operators	non-member

特殊操作符

- 下标运算符[]
  - 必须是成员函数并且只能接受一个参数
  - 通常将返回值定义为引用类型这样就能写如下形式的代码：
    - course OOP;
    - courses[‘oop’] = OOP;
- new/delete运算符
  - 在下一章(第13章)讨论

- 5。22

其它操作符

- Operator ->
  - 当希望一个对象表现得像一个指针时，就需要用到该操作符，有时又称为smart pointer。
  - 该操作符有一个限制：它必须返回一个指针，或者定义了 ->运算符的对象。
  - 一定是成员函数
- 示例：
  - C12:SmartPointer.cpp
  - C12:NestedSmartPointer.cpp

重载赋值运算符

- 等号与构造函数的关系
  - 示例：
    - C12:CopyingVsInitialization.cpp ←
  - 在任何时候，构造一个对象的同时，如果使用一个”=”来初始化一个对象，无论等号右侧是什么，编译器都会寻找一个接受右边参数类型的构造函数对左边的值进行构造。

等号与构造函数的关系

```
class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

int main() {
  Fee fee = 1; // Fee(int)
  Fi fi;
  Fee fum = fi; // Fee(Fi)
} ///:~
```

```
Fee fee(1);
Fee fum(fi);
```

重载赋值运算符

- 等号与构造函数的关系
  - 对于下面的代码：

```
MyType b;
MyType a = b;
a = b;
```

  - 执行第三行a=b时，会发生什么？
    - 调用operator =

重载赋值运算符

- 示例：
  - C12:SimpleAssignment.cpp

重载赋值运算符

- 自动创建operator=
  - 如果用户没有定义operator=，编译器将自动创建一个。
  - 这个操作符的行为与自动创建的拷贝构造函数的行为类似
  - 示例：
    - C12:AtomaticOperatorEquals.cpp

重载赋值运算符

```
class Cargo {
public:
  Cargo& operator=(const Cargo&) {
    cout << "inside Cargo::operator=()" << endl;
    return *this;
  }
};

class Truck {
  Cargo b;
};

int main() {
  Truck a, b;
  a = b; // Prints: "inside Cargo::operator=()"
} ///:~
```

自动类型转换

- 如果编译器看到一个表达式或者函数使用了一个不合适的类型，在可能的情况下（不丢失精度）将会执行一个自动类型转换。
- 如果希望重新定义类型转换的方式，可以通过定义自动类型转换函数来为用户定义类型达到相同效果：特定的构造函数，或者重载运算符。

构造函数转换

- 示例：

```
//: C12:AutomaticTypeConversion.cpp
// Type conversion constructor
class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
  f(one); // Wants a Two, has a One
} ///:~
```

构造函数转换

- 示例：
  - C12:AutomaticTypeConversion.cpp
  - 自动调用构造函数Two::Two(One)
  - 执行时需要额外的开销
- 可以阻止通过构造函数进行的转换
  - C12:ExplicitKeyWork.cpp
  - 必须显式声明转换

构造函数转换

```
//: C12:ExplicitKeyword.cpp
// Using the "explicit" keyword
class One {
public:
  One() {}
};

class Two {
public:
  explicit Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
//!  f(one); // No auto conversion allowed
  f(Two(one)); // OK -- user performs conversion
} ///:~
```

运算符转换

- 第二种自动类型转换的方法是通过运算符重载，创建一个成员函数。通过在关键字operator后跟随想要转换到的类型的方法，实现类型转换。
  - C12: OperatorOverloadingConversion.cpp
  - C12:String1.cpp
  - C12:String2.cpp

自动类型转换潜在的问题

- 由于有多种类型转换的方式，可能存在冲突
  - C12:TypeConversionAmbiguity.cpp
- 存在多个类到同一种类型的转换时，发生冲突
  - C12:TypeConversionFanout.cpp

```
//: C12:TypeConversionAmbiguity.cpp
class Orange; // Class declaration

class Apple {
public:
  operator Orange() const; // Convert Apple to Orange
};

class Orange {
public:
  Orange(Apple); // Convert Apple to Orange
};

void f(Orange) {}

int main() {
  Apple a;
//! f(a); // Error: ambiguous conversion
} ///:~
```

```
//: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
  operator Orange() const;
  operator Pear() const;
};

// Overloaded eat():
void eat(Orange);
void eat(Pear);

int main() {
  Apple c;
//! eat(c);
  // Error: Apple -> Orange or Apple -> Pear ???
} ///:~
```

小结

- C++中，可以为类定义一个运算符，称为运算符重载
- 恰当地使用运算符，可以简化代码的书写，改善代码的可读性。
- 运算符重载的语法
  - 参数个数、类型，返回值类型，成员运算符和非成员运算符
- 一些需特别注意的操作符重载
  - 有无副作用
  - 前后缀表达式
  - 赋值运算
  - 逻辑等于
  - 自动类型转换